Exhibit B

In-App Purchase Programming Guide

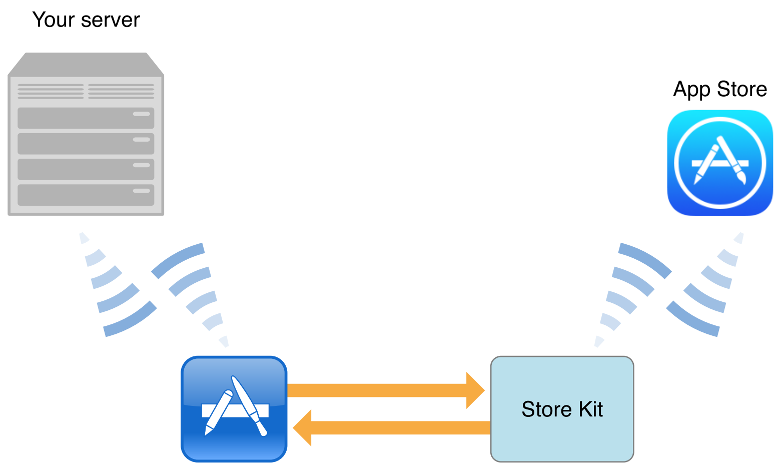
Developer

Contents

Figures, Tables, and Listings

About In-App Purchase

In-App Purchase allows you to embed a store inside your app using the Store Kit framework. This framework connects to the App Store on your app's behalf to securely process payments from users, prompting them to authorize payment. The framework then notifies your app, which provides the purchased items to users. Use In-App Purchase to collect payment for additional features and content.



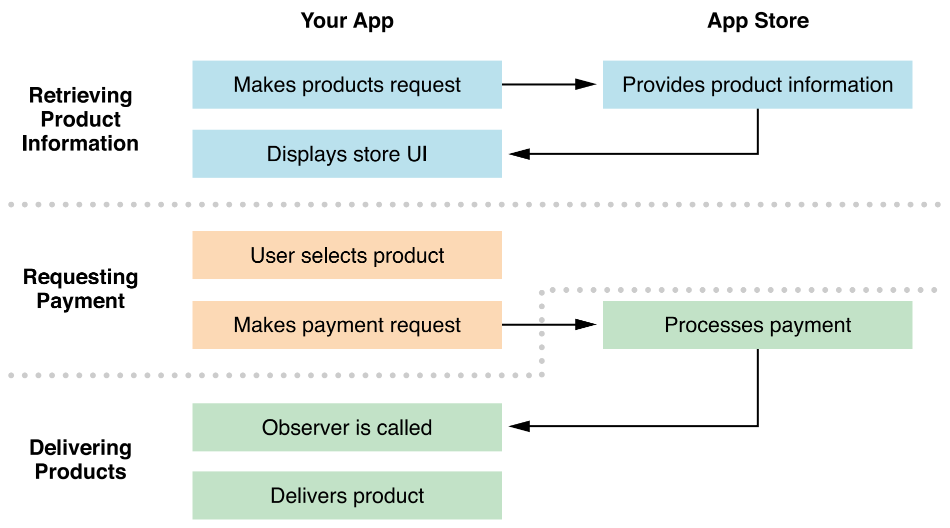
For example, using In-App Purchase, you can implement the following scenarios:

- A basic version of your app with additional premium features
- A magazine app that lets users purchase and download new issues
- A game that offers new levels to explore
- An online game that allows players to purchase virtual property

At a Glance

At a high level, the interactions between the user, your app, and the App Store during the In-App Purchase process take place in three stages, as shown in Figure I-1. First, the user navigates to your app's store and your app displays its products. Second, the user selects a product to buy and the app requests payment from the App Store. Third, the App Store processes the payment and your app delivers the purchased product.

Figure I-1 Stages of the purchase process



You Create and Configure Products in iTunes Connect

Understanding what kinds of products and behaviors are supported by In-App Purchase lets you design your app and in-app store to make the best use of this technology.

Relevant Chapter: "Designing Your App's Products" (page 8)

Your App Interacts with the App Store to Sell Products

All apps that use In-App Purchase need to implement the core functionality described in these chapters to let users make purchases and then deliver the purchased products.

These development tasks need to be done in order. The relevant chapters introduce them in the order you implement them, and they're listed in full in "Implementation Checklist" (page 42). To help plan your development, you may want to read the full checklist before you begin.

Relevant Chapters: "Retrieving Product Information" (page 12), "Requesting Payment" (page 20), "Delivering Products" (page 23)

Subscriptions Require Additional Application Logic

Apps that offer subscriptions need to keep track of when the user has an active subscription, respond to expiration and renewal, and determine what content the user has access to.

Relevant Chapter: "Working with Subscriptions" (page 34)

Users Can Restore Purchases

Users can restore products that they previously purchased—for example, to bring content they've already paid for onto their new phone.

Relevant Chapter: "Restoring Purchased Products" (page 38)

Apps and Products Are Submitted for Review

When you're done developing and testing, you submit your app and your In-App Purchase products for review.

Relevant Chapter: "Preparing for App Review" (page 41)

See Also

- "In-App Purchase" in *iTunes Connect Developer Guide* describes how to use iTunes Connect, with a focus on how to create and configure your app's products.
- *Receipt Validation Programming Guide* describes how to work with receipts, in particular with the record of successful in-app purchases.

Designing Your App's Products

A **product** is something you want to sell in your app's store. You create and configure products in iTunes Connect, and your app interacts with products using the `SKProduct` and `SKProductsRequest` classes.

Understanding What You Can Sell Using In-App Purchase

You can use In-App Purchase to sell content, app functionality, and services.

- **Content.** Deliver digital content or assets, such as magazines, photos, and artwork. Content can also be used by the app itself—for example, additional characters and levels in a game, filters in a camera app, and stationery in a word processor.
- **App functionality.** Unlock behavior and expand features you've already delivered. Examples include a free game that offers multiplayer mode as an in-app purchase and a free weather app that lets users make a one-time purchase to remove ads.
- **Services.** Have users pay for one-time services such as voice transcription and for ongoing services such as access to a collection of data.

You *can't* use In-App Purchase to sell real-world goods and services or to sell unsuitable content.

- **Real-world goods and services.** You must deliver a digital good or service within your app when using In-App Purchase. Use a different payment mechanism to let your users buy real-world goods and services in your app, such as a credit card or payment service.
- **Unsuitable content.** Don't use In-App Purchase to sell content that the isn't allowed by the App Review Guidelines—for example, pornography, hate speech, or defamation.

For detailed information about what you can offer using In-App Purchase, see your license agreement and the App Review Guidelines. Reviewing the guidelines carefully before you start coding helps you avoid delays and rejection during the review process. If the guidelines don't address your case in sufficient detail, you can ask the App Review team specific questions using the online contact form.

After you know what products you want to sell in your app and determine that In-App Purchase is the appropriate way to sell those products, you need to create the products in iTunes Connect.

Creating Products in iTunes Connect

Before you start coding, you need to configure products in iTunes Connect for your app to interact with. As you develop your app, you can add and remove products and refine or reconfigure your existing products.

Products are reviewed when you submit your app as part of the app review process. Before users can buy a product, it must be approved by the reviewer and you must mark it as "cleared for sale" in iTunes Connect.

For details about configuring products in iTunes Connect, see "In-App Purchase" in *iTunes Connect Developer Guide* .

Product Types

Product types let you use In-App Purchase in a range of apps by providing several different product behaviors. In iTunes Connect, you select one of the following product types:

- **Consumable products.** Items that get used up over the course of running your app. Examples include minutes for a Voice over IP app and one-time services such as voice transcription.
- **Non-consumable products.** Items that remain available to the user indefinitely on all of the user's devices. They're made available to all of the user's devices. Examples include content, such as books and game levels, and additional app functionality.
- **Auto-renewable subscriptions.** Episodic content. Like non-consumable products, auto-renewable subscriptions remain available to the user indefinitely on all of the user's devices. Unlike non-consumable products, auto-renewable subscriptions have an expiration date. You deliver new content regularly, and users get access to content published during the time period their subscription is active. When an auto-renewable subscription is about to expire, the system automatically renews it on the user's behalf.
- **Non-renewable subscriptions.** Subscriptions that don't involve delivering episodic content. Examples include access to a database of historic photos or a collection of flight maps. It's your app's responsibility to make the subscription available on all of the user's devices and to let users restore the purchase. This product type is often used when your users already have an account on your server that you can use to identify them when restoring content. Expiration and the duration of the subscription are also left to your app (or your server) to implement and enforce.
- **Free subscriptions.** A way to put free subscription content in Newsstand. After a user signs up for a free subscription, the content is available on all devices associated with the user's Apple ID. Free subscriptions don't expire and can be offered only in Newsstand-enabled apps.

Differences Between Product Types

Each product type is designed for a particular use. The behavior of different product types varies in certain ways, as summarized in Table 1-1 and Table 1-2.

Table 1-1 Comparison of product types

Product type	Non-consumable	Consumable
Users can buy	Once	Multiple times
Appears in the receipt	Always	Once
Synced across devices	By the system	Not synced
Restored	By the system	Not restored

Table 1-2 Comparison of subscription types

Subscription type	Auto-renewable	Non-renewing	Free
Users can buy	Multiple times	Multiple times	Once
Appears in the receipt	Always	Once	Always
Synced across devices	By the system	By your app	By the system
Restored	By the system	By your app	By the system

Products that expire or get used up—consumable products, auto-renewable subscriptions, and non-renewing subscriptions—can be purchased multiple times to get the consumable item again or extend the subscription. Non-consumable products and free subscriptions unlock content that remains available to the user indefinitely, so these can only be purchased once.

Consumable products and free subscriptions appear in the receipt after being purchased but are removed the next time the receipt is updated, as discussed in more detail in "Persisting Using the App Receipt" (page 26). All other types of products have an entry in the receipt that isn't removed.

Consumable products, by their nature, aren't synced or restored. Users understand that, for example, buying ten more bubbles on their iPhone doesn't also give them ten more bubbles on their iPad. All other types of products are made available across all of the user's devices. They're also restored so users can continue to access their purchased content even after buying a new device. Store Kit handles the syncing and restoring process for auto-renewable and free subscriptions and for non-consumable products.

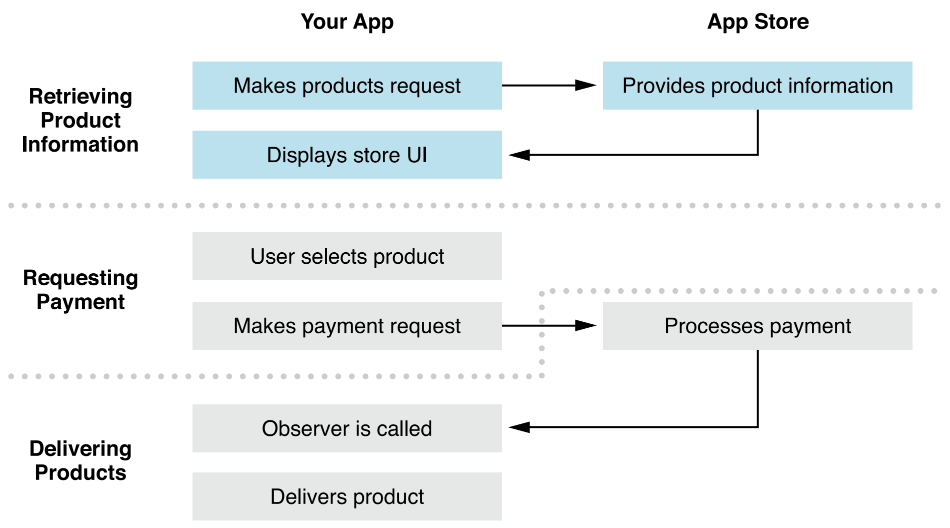
Non-renewing subscriptions differ from auto-renewable subscriptions in a few key ways. These differences give your app the flexibility to implement the correct behavior for your needs, as follows:

- Your app is responsible for calculating the time period that the subscription is active and determining what content needs to be made available to the user.
- Your app is responsible for detecting that a subscription is approaching its expiration date and prompting the user to renew the subscription by purchasing the product again.
- Your app is responsible for making subscriptions available across all the user's devices after they're purchased and for letting users restore past purchases. For example, most subscriptions are provided by a server; your server would need some mechanism to identify users and associate subscription purchases with the user who purchased them.

Retrieving Product Information

In the first part of the purchase process, your app retrieves information about its products from the App Store, presents its store UI to the user, and then lets the user select a product, as shown in Figure 2-1.

Figure 2-1 Stages of the purchase process—displaying store UI



Getting a List of Product Identifiers

Every product you sell in your app has a unique **product identifier**. Your app uses these product identifiers to fetch information about products from the App Store, such as pricing, and to submit payment requests when users purchase those products. Your app can either read its list of product identifiers from a file in its app bundle or fetch them from your server. Table 2-1 summarizes the differences between the two approaches.

Table 2-1 Comparison of approaches for obtaining product identifiers

	Embedded in the app bundle	Fetched from your server
Used for purchases that	Unlock functionality	Deliver content
List of products can change	When the app is updated	At any time
Requires a server	No	Yes

If your app has a fixed list of products, such as an in-app purchase to remove ads or enable functionality, embed the list in the app bundle. If the list of product identifiers can change without your app needing to be updated, such as a game that supports additional levels or characters, have your app fetch the list from your server.

There's no runtime mechanism to fetch a list of all products configured in iTunes Connect for a particular app. You're responsible for managing your app's list of products and providing that information to your app. If you need to manage a large number of products, consider using the bulk XML upload/download feature in iTunes Connect.

Embedding Product IDs in the App Bundle

Include a property list file in your app bundle containing an array of product identifiers, such as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <string>com.example.level1</string>
    <string>com.example.level2</string>
    <string>com.example.rocket_car</string>
</array>
</plist>
```

To get product identifiers from the property list, locate the file in the app bundle and read it.

```
NSURL *url = [[NSBundle mainBundle] URLForResource:@"product_ids"
                                     withExtension:@"plist"];
NSArray *productIdentifiers = [NSArray arrayWithContentsOfURL:url];
```

Fetching Product IDs from Your Server

Host a JSON file on your server with the product identifiers. For example:

```
[
    "com.example.level1",
    "com.example.level2",
    "com.example.rocket_car"
]
```

To get product identifiers from your server, fetch and read the JSON file as shown in Listing 2-1. Consider versioning the JSON file so that future versions of your app can change its structure without breaking older versions of your app. For example, you could name the file that uses the old structure `products_v1.json` and the file that uses a new structure `products_v2.json`. This is especially useful if your JSON file is more complex than the simple array in the example.

Listing 2-1 Fetching product identifiers from your server

```
-fetchProductIdentifiersFromURL:(NSURL *)url delegate(id):delegate
{
    dispatch_queue_t global_queue =
           dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(global_queue, ^{
        NSError *err;
        NSData *jsonData = [NSData dataWithContentsOfURL:url
                                                  options:NULL
                                                    error:&err];
        if (!jsonData) { /* Handle the error */ }

        NSArray *productIdentifiers = [NSJSONSerialization
            JSONObjectWithData:jsonData options:NULL error:&err];
        if (!productIdentifiers) { /* Handle the error */ }

        dispatch_queue_t main_queue = dispatch_get_main_queue();
        dispatch_async(main_queue, ^{
           [delegate displayProducts:productIdentifiers]; // Custom method
        }
    });
}
```

For information about downloading files using `NSURLConnection`, see “Using NSURLConnection” in *URL Loading System Programming Guide*.

To ensure that your app remains responsive, use a background thread to download the JSON file and extract the list of product identifiers. To minimize the data transferred, use standard HTTP caching mechanisms, such as the `Last-Modified` and `If-Modified-Since` headers.

Retrieving Product Information

To make sure your users see only products that are actually available for purchase, query the App Store before displaying your app's store UI.

Use a products request object to query the App Store. First, create an instance of `SKProductsRequest` and initialize it with a list of product identifiers. The products request retrieves information about valid products, along with a list of the invalid product identifiers, and then calls its delegate to process the result. The delegate must implement the `SKProductsRequestDelegate` protocol to handle the response from the App Store. Listing 2-2 shows a simple implementation of both pieces of code.

Listing 2-2 Retrieving product information

```
// Custom method
- validateProductIdentifiers:(NSArray *)productIdentifiers
{
    SKProductsRequest *productsRequest = [[SKProductsRequest alloc]
        initWithProductIdentifiers:[NSSet setWithArray:productIdentifiers]];
    productsRequest.delegate = self;
    [productsRequest start];
}

// SKProductsRequestDelegate protocol method
- (void)productsRequest:(SKProductsRequest *)request
     didReceiveResponse:(SKProductsResponse *)response
{
    self.products = response.products;

    for (NSString *invalidIdentifier in response.invalidIdentifiers) {
        // Handle any invalid product identifiers.
    }

    [self displayStoreUI]; // Custom method
}
```

When the user purchases a product, you need the corresponding product object to create a payment request, so keep a reference to the array of product objects that's returned to the delegate. If the list of products your app sells can change, you may want to create a custom class that encapsulates a reference to the product object as well as other information—for example, pictures or description text that you fetch from your server. Payment requests are discussed in "Requesting Payment" (page 20).

Product identifiers being returned as invalid usually indicates an error in your app's list of product identifiers, although it could mean the product hasn't been properly configured in iTunes Connect. Good logging and a good UI help you resolve this type of issue more easily. In production builds, your app needs to fail gracefully—typically, this means displaying the rest of your app's store UI and omitting the invalid product. In development builds, display an error to call attention to the issue. In both production and development builds, use `NSLog` to write a message to the console so you have a record of the invalid identifier. If your app fetched the list from your server, you could also define a logging mechanism to let your app send the list of invalid identifiers back to your server.

Presenting Your App's Store UI

Because the design of your app's store has an important impact on your in-app sales, it's worth investing the time and effort to get it right. Design the user interface for your store UI so it integrates with the rest of your app. Store Kit can't provide a store UI for you. Only you know your app and its content well enough to design your store UI in a way that showcases your products in their best light and fits seamlessly with the rest of your app.

Consider the following guidelines as you design and implement your app's store UI.

Display a store only if the user can make payments. To determine whether the user can make payments, call the `canMakePayments` class method of the `SKPaymentQueue` class. If the user can't make payments (for example, because of parental restrictions), either display UI indicating that that the store isn't available or omit the store portion of your UI entirely.

Present products naturally in the flow of your app. Find the best place in your UI to show your app's store UI. Present products in context at the time when the user can use them—for example, let users unlock functionality when they try to use that premium feature. Pay special attention to the experience a user has when exploring your app for the first time.

Organize products so that exploration is easy and enjoyable. If your app has a small enough number of products, you can display everything on one screen; otherwise, group or categorize products to make them easy to navigate. Apps with a large number of products, such as comic book readers or magazines with many issues, benefit especially from an interface that makes it easy for users to discover new items they want to purchase. Make the differences between your products clear by giving them distinct names and visuals—if necessary, include explicit comparisons.

Communicate the value of your products to your users. Users want to know exactly what they're going to buy. Combine information from the App Store, such as product prices and descriptions, with additional data from your server or the app bundle, such as images or demos of your products. Let users interact with a product in a limited way before buying it. For example, a game that gives the user the option to buy new race cars can allow users to run a test lap with the new car. Likewise, a drawing app that lets the user buy additional brushes can give users the chance to draw with the new brush on a small scratch pad and see the difference between brushes. This kind of design provides users an opportunity to experience the product and be convinced they want to purchase it.

Display prices clearly, using the locale and currency returned by the App Store. Ensure that the price of a product is easy to find and easy to read. Don't try to convert the price to a different currency in your UI, even if the user's locale and the price's locale differ. Consider, for example, a user in the United States who prefers the United Kingdom locale for its units and date formatting. Your app displays its UI according to the United Kingdom locale, but it still needs to display product information in the locale specified by the App Store. Converting prices to British pounds sterling, in an attempt to match the United Kingdom locale of the rest of the interface, would be incorrect. The user has an App Store account in the United States and pays in U.S. dollars, so prices would be provided to your app in U.S. dollars. Likewise, your app would display its prices in U.S. dollars. Listing 2-3 shows how to correctly format a price by using the product's locale information.

Listing 2-3 Formatting a product's price

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
[numberFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
[numberFormatter setLocale:product.priceLocale];
NSString *formattedPrice = [numberFormatter stringFromNumber:product.price];
```

After a user selects a product to buy, your app connects to the App Store to request payment for the product.

Suggested Testing Steps

Test each part of your code to verify that you've implemented it correctly.

Sign In to the App Store with Your Test Account

Create a test user account in iTunes Connect, as described in “In-App Purchase” in *iTunes Connect Developer Guide*.

On a development iOS device, sign out of the App Store in Settings. Then build and run your app from Xcode.

On a development OS X device, sign out of the Mac App Store. Then build your app in Xcode and launch it from the Finder.

Use your app to make an in-app purchase. When prompted to sign in to the App Store, use your test account. Note that the text “[Environment: Sandbox]” appears as part of the prompt, indicating that you’re connected to the test environment.

If the text “[Environment: Sandbox]” doesn’t appear, you’re using the production environment. Make sure you’re running a development-signed build of your app. Production-signed builds use the production environment.

Important: Don’t use your test user account to sign in to the production environment. If you do, the test user account becomes invalid and can no longer be used.

Test Fetching the List of Product Identifiers

If your product identifiers are embedded in your app, set a breakpoint in your code after they’re loaded and verify that the instance of `NSArray` contains the expected list of product identifiers.

If your product identifiers are fetched from a server, manually fetch the JSON file—using a web browser such as Safari or a command-line utility such as `curl`—and verify that the data returned from your server contains the expected list of product identifiers. Also verify that your server correctly implements standard HTTP caching mechanisms.

Test Handling of Invalid Product Identifiers

Intentionally include an invalid identifier in your app’s list of product identifiers. (Make sure you remove it after testing.)

In a production build, verify that the app displays the rest of its store UI and users can purchase other products. In a development build, verify that the app brings the issue to your attention.

Check the console log and verify that you can correctly identify the invalid product identifier.

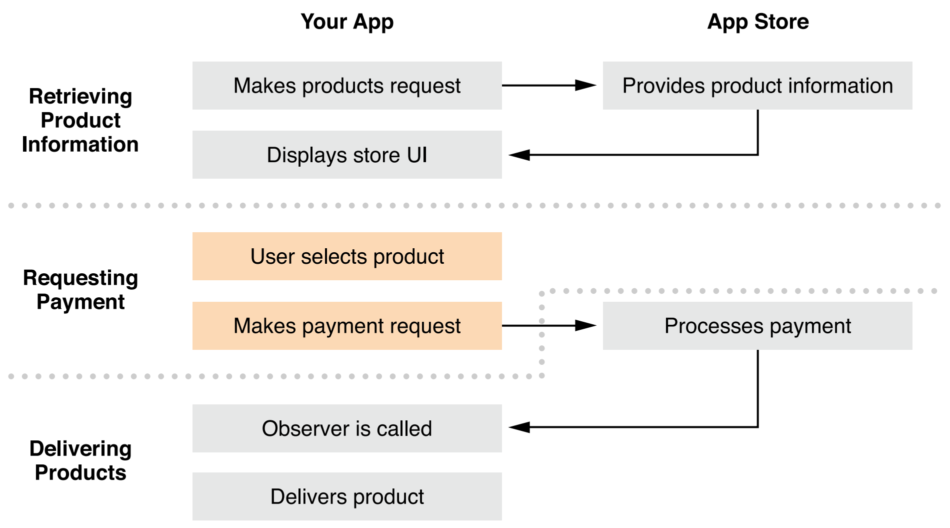
Test a Products Request

Using the list of product identifiers that you tested, create and submit an instance of `SKProductsRequest`. Set a breakpoint in your code, and inspect the lists of valid and invalid product identifiers. If there are invalid product identifiers, review your products in iTunes Connect and correct your JSON file or property list.

Requesting Payment

In the second part of the purchase process, after the user has chosen to purchase a particular product, your app submits a payment request to the App Store, as shown in Figure 3-1.

Figure 3-1 Stages of the purchase process—requesting payment



Creating a Payment Request

When the user selects a product to buy, create a payment request using a product object, and set the quantity if needed, as shown in Listing 3-1. The product object comes from the array of products returned by your app's products request, as discussed in "Retrieving Product Information" (page 15).

Listing 3-1 Creating a payment request

```
SKProduct *product = <# Product returned by a products request #>;
SKMutablePayment *payment = [SKMutablePayment paymentWithProduct:product];
payment.quantity = 2;
```

Detecting Irregular Activity

The App Store uses an irregular activity detection engine to help combat fraud. Some apps can provide additional information to improve the engine's ability to detect unusual transactions. If your users have an account with you, in addition to their App Store accounts, provide this additional piece of information when requesting payment.

By way of illustration, consider the following two examples. In the normal case, many different users on your server each buy coins to use in your game, and each user pays for the purchase from a different App Store account. In contrast, it would be very unusual for a single user on your server to buy coins multiple times, paying for each purchase from a different App Store account. The App Store can't detect this kind of irregular activity on its own—it needs information from your app about which account on your server is associated with the transaction.

To provide this information, populate the `applicationUsername` property of the payment object with a one-way hash of the user's account name on your server, such as in the example shown in Listing 3-2.

Listing 3-2 Providing an application username

```
#import <CommonCrypto/CommonCrypto.h>

// Custom method to calculate the SHA-256 hash using Common Crypto
- (NSString *)hashedValueForAccountName:(NSString*)userAccountName
{
    const int HASH_SIZE = 32;
    unsigned char hashedChars[HASH_SIZE];
    const char *accountName = [userAccountName UTF8String];
    size_t accountNameLen = strlen(accountName);

    // Confirm that the length of the user name is small enough
    // to be recast when calling the hash function.

    if (accountNameLen > UINT32_MAX) {
        NSLog(@"Account name too long to hash: %@", userAccountName);
        return nil;
    }
    CC_SHA256(accountName, (CC_LONG)accountNameLen, hashedChars);

    // Convert the array of bytes into a string showing its hex representation.
    NSMutableString *userAccountHash = [[NSMutableString alloc] init];
    for (int i = 0; i < HASH_SIZE; i++) {
        // Add a dash every four bytes, for readability.
        if (i != 0 && i%4 == 0) {
            [userAccountHash appendString:@"-"];
        }
        [userAccountHash appendFormat:@"%02x", hashedChars[i]];
    }

    return userAccountHash;
}
```

If you use another approach to populate this property, ensure that the value you provide is an opaque identifier uniquely associated with the user's account on your server. Don't use the Apple ID for your developer account, the user's Apple ID, or the user's unhashed account name on your server.

Submitting a Payment Request

Adding a payment request to the transaction queue submits it the App Store. If you add a payment object to the queue multiple times, it's submitted multiple times—the user is charged multiple times and your app is expected to deliver the product multiple times.

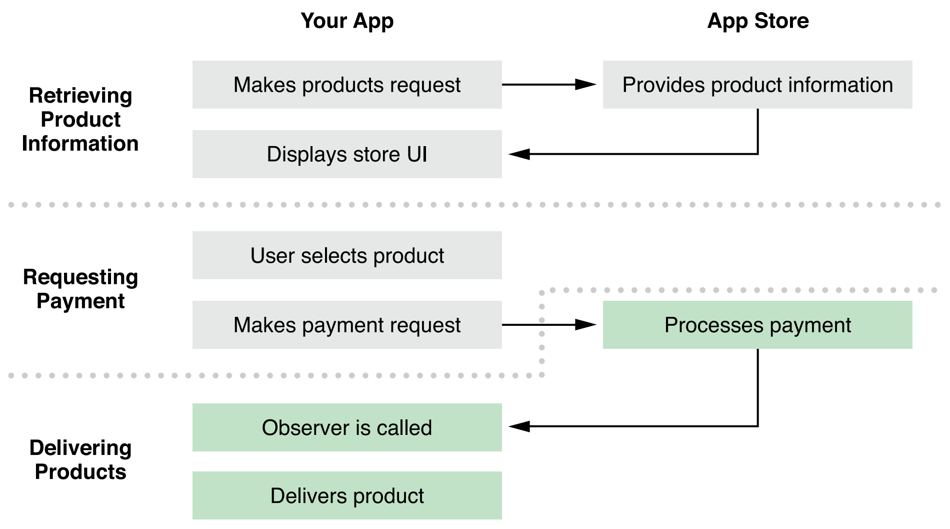
```
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

For every payment request your app submits, it gets back a corresponding transaction that it must process. Transactions and the transaction queue are discussed in "Waiting for the App Store to Process Transactions" (page 23).

Delivering Products

In the final part of the purchase process, your app waits for the App Store to process the payment request, stores information about the purchase for future launches, downloads the purchased content, and then marks the transaction as finished, as shown in Figure 4-1.

Figure 4-1 Stages of the purchase process—delivering products



Waiting for the App Store to Process Transactions

The **transaction queue** plays a central role in letting your app communicate with the App Store through the Store Kit framework. You add work to the queue that the App Store needs to act on, such as a payment request that needs to be processed. When the transaction's state changes—for example, when a payment request succeeds—Store Kit calls the app's **transaction queue observer**. Using an observer this way means your app doesn't constantly poll the status of its active transactions. In addition to using the transaction queue for payment requests, your app also uses the transaction queue to download Apple-hosted content and to find out that subscriptions have been renewed.

Register a transaction queue observer when your app is launched, as shown in Listing 4-1. Make sure that the observer is ready to handle a transaction at any time, not just after you add a transaction to the queue. For example, consider the case of a user buying something in your app right before going into a tunnel. Your app isn't able to deliver the purchased content because there's no network connection. The next time your app is launched, Store Kit calls your transaction queue observer again and delivers the purchased content at that time. Similarly, if your app fails to mark a transaction as finished, Store Kit calls the observer every time your app is launched until the transaction is properly finished.

Listing 4-1 Registering the transaction queue observer

```
- (BOOL)application:(UIApplication *)application
 didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    /* ... */

    [[SKPaymentQueue defaultQueue] addTransactionObserver:observer];
}
```

Implement the `paymentQueue:updatedTransactions:` method on your transaction queue observer. Store Kit calls this method when the status of a transaction changes—for example, when a payment request has been processed. The transaction status tells you what action your app needs to perform, as shown in Table 4-1 and Listing 4-2.

Table 4-1 Transaction statuses and corresponding actions

Status	Action to take in your app
`SKPaymentTransactionStatePurchasing`	Update your UI to reflect the status, and wait to be called again.
`SKPaymentTransactionStateFailed`	Use the value of the `error` property to present a message to the user.
`SKPaymentTransactionStatePurchased`	Provide the purchased functionality.
`SKPaymentTransactionStateRestored`	Restore the previously purchased functionality.

Listing 4-2 Responding to transaction statuses

```
- (void)paymentQueue:(SKPaymentQueue *)queue
 updatedTransactions:(NSArray *)transactions
{
    for (SKPaymentTransaction *transaction in transactions) {
        switch (transaction.transactionState) {
            // Call the appropriate custom method.
            case SKPaymentTransactionStatePurchased:
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored:
                [self restoreTransaction:transaction];
            default:
                break;
        }
    }
}
```

To keep your user interface up to date while waiting, the transaction queue observer can implement optional methods from the `SKPaymentTransactionObserver` protocol as follows. The `paymentQueue:removedTransactions:` method is called when transactions are removed from the queue—in your implementation of this method, remove the corresponding items from your app's UI. The `paymentQueueRestoreCompletedTransactionsFinished:` or `paymentQueue:restoreCompletedTransactionsFailedWithError:` method is called when Store Kit finishes restoring transactions, depending on whether there was an error. In your implementation of these methods, update your app's UI to reflect the success or error.

Persisting the Purchase

After making the product available, your app needs to make a persistent record of the purchase. Your app uses that persistent record on launch to continue to make the product available. It also uses that record to restore purchases, as described in "Restoring Purchased Products" (page 38). Your app's persistence strategy depends the type of products you sell and the versions of iOS.

- For non-consumable products and auto-renewable subscriptions in iOS 7 and later, use the app receipt as your persistent record.
- For non-consumable products and auto-renewable subscriptions in versions of iOS earlier than iOS 7, use the User Defaults system or iCloud to keep a persistent record.
- For non-renewing subscriptions, use iCloud or your own server to keep a persistent record.

- For consumable products, your app updates its internal state to reflect the purchase, but there's no need to keep a persistent record because consumable products aren't restored or synced across devices. Ensure that the updated state is part of an object that supports state preservation (in iOS) or that you manually preserve the state across app launches (in iOS or OS X). For information about state preservation, see "State Preservation and Restoration" in *iOS App Programming Guide* .

When using the User Defaults system or iCloud, your app can store a value, such as a number or a Boolean, or a copy of the transaction receipt. In OS X, the user can edit the User Defaults system using the `defaults` command. Storing a receipt requires more application logic, but prevents the persistent record from being tampered with.

When persisting via iCloud, note that your app's persistent record is synced across devices, but your app is responsible for downloading any associated content on other devices.

Persisting Using the App Receipt

The app receipt contains a record of the user's purchases, cryptographically signed by Apple. For more information, see *Receipt Validation Programming Guide* .

Information about consumable products and non-renewing subscriptions is added to the receipt when they're paid for and remains in the receipt until you finish the transaction. After you finish the transaction, this information is removed the next time the receipt is updated—for example, the next time the user makes a purchase.

Information about all other kinds of purchases is added to the receipt when they're paid for and remains in the receipt indefinitely.

Persisting a Value in User Defaults or iCloud

To store information in User Defaults or iCloud, set the value for a key.

```
#if USE_ICLOUD_STORAGE
NSUbiquitousKeyValueStore *storage = [NSUbiquitousKeyValueStore defaultStore];
#else
NSUserDefaults *storage = [NSUserDefaults standardUserDefaults];
#endif

[storage setBool:YES forKey:@"enable_rocket_car"];
[storage setObject:@15 forKey:@"highest_unlocked_level"];
```

```
[storage synchronize];
```

Persisting a Receipt in User Defaults or iCloud

To store a transaction's receipt in User Defaults or iCloud, set the value for a key to the data of that receipt.

```
#if USE_ICLOUD_STORAGE
NSUbiquitousKeyValueStore *storage = [NSUbiquitousKeyValueStore defaultStore];
#else
NSUserDefaults *storage = [NSUserDefaults standardUserDefaults];
#endif

NSData *newReceipt = transaction.transactionReceipt;
NSArray *savedReceipts = [storage arrayForKey:@"receipts"];
if (!receipts) {
    // Storing the first receipt
    [defaults setObject:@[receipt] forKey@"receipts"];
} else {
    // Adding another receipt
    NSArray *updatedReceipts = [savedReceipts arrayByAddingObject:newReceipt];
    [defaults setObject:updatedReceipts forKey@"receipts"];
}

[storage synchronize];
```

Persisting Using Your Own Server

Send a copy of the receipt to your server along with some kind of credentials or identifier so you can keep track of which receipts belong to a particular user. For example, let users identify themselves to your server with an email or user name, plus a password. Don't use the `identifierForVendor` property of `UIDevice`—you can't use it to identify and restore purchases made by the same user on a different device, because different devices have different values for this property.

Unlocking App Functionality

If the product enables app functionality, set a Boolean value to enable the code path and update your user interface as needed. To determine what functionality to unlock, consult the persistent record that your app made when the transaction occurred. Your app needs to update this Boolean value whenever a purchase is completed and at app launch.

For example, using the app receipt, your code might look like the following:

```
NSURL *receiptURL = [[NSBundle mainBundle] appStoreReceiptURL];
NSData *receiptData = [NSData dataWithContentsOfURL:receiptURL];

// Custom method to work with receipts
BOOL rocketCarEnabled = [self receipt:receiptData
        includesProductID:@"com.example.rocketCar"];
```

Or, using the User Defaults system:

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
BOOL rocketCarEnabled = [defaults boolForKey:@"enable_rocket_car"];
```

Then use that information to enable the appropriate code paths in your app.

```
if (rocketCarEnabled) {
    // Use the rocket car.
} else {
    // Use the regular car.
}
```

Delivering Associated Content

If the product has associated content, your app needs to deliver that content to the user. For example, purchasing a level in a game requires delivering the files that define that level, and purchasing additional instruments in a music app requires delivering the sound files needed to let the user play those instruments.

You can embed that content in your app's bundle or you can download it as needed—each approach has its advantages and disadvantages. If you include too little content in your app bundle, the user must wait for even small purchases to be downloaded. If you include too much in your app bundle, the initial download of the app takes too long, and the space is wasted for users who don't purchase the corresponding products. Additionally, if your app is too large, users won't be able to download it over cellular networks.

Embed smaller files (up to a few megabytes) in your app, especially if you expect most users to buy that product. Content in your app bundle can be made available immediately when the user purchases it. However, to add or update content in your app bundle, you have to submit an updated version of your app.

Download larger files when needed. Separating content from your app bundle keeps your app's initial download small. For example, a game can include the first level in its app bundle and let users download the rest of the levels when they're purchased. Assuming your app fetches its list of product identifiers from your server, and not hard-coded in the app bundle, you don't need to resubmit your app to add or update content that is downloaded by your app.

In iOS 6 and later, most apps should use Apple-hosted content for downloaded files. You create an Apple-hosted content bundle using the In-App Purchase Content target in Xcode and submit it to iTunes Connect. When you host content on Apple's servers you don't need to provide any servers—your app's content is stored by Apple using the same infrastructure that supports other large-scale operations, such as the App Store. Additionally, Apple-hosted content automatically downloads in the background even if your app isn't running.

You might choose to host your own content if you already have server infrastructure, if you need to support older versions of iOS, or if you share your server infrastructure across multiple platforms.

Note: You can't patch your app binary or download executable code. Your app must contain all executable code needed to support all of its functionality when you submit it. If a new product requires code changes, submit an updated version of your app.

Loading Local Content

Load local content using the `NSBundle` class, just as you load other resources from your app bundle.

```
NSURL *url = [[NSBundle mainBundle] URLForResource:@"rocketCar"
                                     withExtension:@"plist"];
[self loadVehicleAtURL:url];
```

Downloading Hosted Content from Apple's Server

When the user purchases a product that has associated Apple-hosted content, the transaction passed to your transaction queue observer also includes an instance of `SKDownload` that lets you download the associated content.

To download the content, add the download objects from the transaction's `downloads` property to the transaction queue by calling the `startDownloads:` method of `SKPaymentQueue`. If the value of the `downloads` property is `nil`, there's no Apple-hosted content for that transaction. Unlike downloading apps, downloading content doesn't automatically require a Wi-Fi connection for content larger than a certain size. Avoid using cellular networks to download large files without an explicit action from the user.

Implement the `paymentQueue:updatedDownloads:` method on the transaction queue observer to respond to changes in a download's state—for example, by updating progress in your UI. If a download fails, use the information in its `error` property to present the error to the user.

Ensure that your app handles errors gracefully. For example, if the device runs out of disk space during a download, give the user the option to discard the partial download or to resume the download later when space becomes available.

Update your user interface while the content is downloading using the values of the `progress` and `timeRemaining` properties. You can use the `pauseDownloads:`, `resumeDownloads:`, and `cancelDownloads:` methods of `SKPaymentQueue` from your UI to let the user control in-progress downloads. Use the `downloadState` property to determine whether the download has completed. Don't use the `progress` or `timeRemaining` property of the download object to check its status—these properties are for updating your UI.

Note: Download all Apple-hosted content before finishing the transaction. After a transaction is complete, its download objects can no longer be used.

In iOS, your app can manage the downloaded files. The files are saved for you by the Store Kit framework in the `Caches` directory with the backup flag unset. After the download completes, your app is responsible for moving it to the appropriate location. For content that can be deleted if the device runs out of disk space (and later re-downloaded by your app), leave the files in the `Caches` directory. Otherwise, move the files to the `Documents` folder and set the flag to exclude them from user backups.

Listing 4-3 Excluding downloaded content from backups

```
NSError *error;
BOOL success = [URL setResourceValue:[NSNumber numberWithBool:YES]
```

```
                                            forKey:NSURLIsExcludedFromBackupKey
                                             error:&error];
if (!success) { /* Handle error... */ }
```

In OS X, the downloaded files are managed by the system; your app can't move or delete them directly. To locate the content after downloading it, use the `contentURL` property of the download object. To locate the file on subsequent launches, use the `contentURLForProductID:` class method of `SKDownload`. To delete a file, use the `deleteContentForProductID:` class method. For information about reading the product identifiers from your app's receipt, see *Receipt Validation Programming Guide* .

Downloading Content from Your Own Server

As with all other interactions between your app and your server, the details and mechanics of the process of downloading content from your own server are your responsibility. The communication consists of, at a minimum, the following steps:

1. Your app sends the receipt to your server and requests the content.
2. Your server validates the receipt to establish that the content has been purchased, as described in *Receipt Validation Programming Guide* .
3. Assuming the receipt is valid, your server responds to your app with the content.

Ensure that your app handles errors gracefully. For example, if the device runs out of disk space during a download, give the user the option to discard the partial download or to resume the download later when space becomes available.

Consider the security implications of how you host your content and of how your app communicates with your server. For more information, see *Security Overview* .

Finishing the Transaction

Finishing a transaction tells Store Kit that you've completed everything needed for the purchase. Unfinished transactions remain in the queue until they're finished, and the transaction queue observer is called every time your app is launched so your app can finish the transactions.

Complete all of the following actions before you finish the transaction:

- Persist the purchase.
- Download associated content.

- Update your app's UI to let the user access the product.

To finish a transaction, call the `finishTransaction:` method on the payment queue.

```
SKPaymentTransaction *transaction = <# The current payment #>;
SKPaymentQueue *queue = [SKPaymentQueue defaultQueue];
[defaultQueue finishTransaction:transaction;]
```

After you finish a transaction, don't take any actions on that transaction or do any work to deliver the product. If any work remains, your app isn't ready to finish the transaction yet.

Note: Don't try to call the `finishTransaction:` method before the transaction is actually completed, attempting to use some other mechanism in your app to track the transaction as unfinished. Store Kit isn't designed to be used this way. Doing so prevents your app from downloading Apple-hosted content and can lead to other issues.

Suggested Testing Steps

Test each part of your code to verify that you've implemented it correctly.

Test a Payment Request

Create an instance of `SKPayment` using a valid product identifier that you've already tested. Set a breakpoint and inspect the payment request. Add the payment request to the transaction queue, and set a breakpoint to confirm that the `paymentQueue:updatedTransactions:` method of your observer is called.

During testing, it's OK to finish the transaction immediately without providing the content. However, even during testing, failing to finish the transaction can cause problems: unfinished transactions remain in the queue indefinitely, which could interfere with later testing.

Verify Your Observer Code

Review the transaction observer's implementation of the `SKPaymentTransactionObserver` protocol. Verify that it can handle transactions even if you aren't currently displaying your app's store UI and even if you didn't recently initiate a purchase.

Locate the call to the `addTransactionObserver:` method of `SKPaymentQueue` in your code. Verify that your app calls this method at app launch.

Test a Successful Transaction

Sign in to the App Store with a test user account, and make a purchase in your app. Set a breakpoint in your implementation of the transaction queue observer's `paymentQueue:updatedTransactions:` method, and inspect the transaction to verify that its status is `SKPaymentTransactionStatePurchased`.

Set a breakpoint at the point in your code that persists the purchase, and confirm that this code is called in response to a successful purchase. Inspect the User Defaults or iCloud key-value store, and confirm that the correct information has been recorded.

Test an Interrupted Transaction

Set a breakpoint in your transaction queue observer's `paymentQueue:updatedTransactions:` method so you can control whether it delivers the product. Then make a purchase as usual in the test environment, and use the breakpoint to temporarily ignore the transaction—for example, by returning from the method immediately using the `thread return` command in LLDB.

Terminate and relaunch your app. Store Kit calls the `paymentQueue:updatedTransactions:` method again shortly after launch; this time, let your app respond normally. Verify that your app correctly delivers the product and completes the transaction.

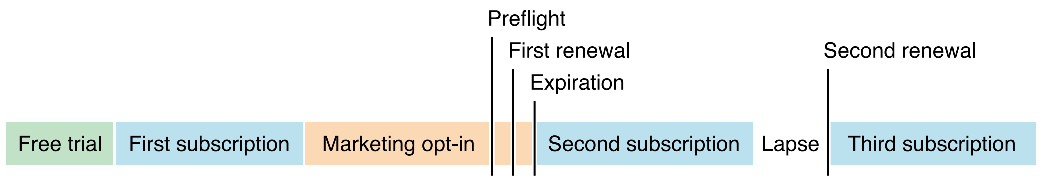
Verify That Transactions Are Finished

Locate where your app calls the `finishTransaction:` method. Verify that all work related to the transaction has been completed before the method is called and that the method is called for every successful purchase.

Working with Subscriptions

Apps that use subscriptions have some additional behaviors and considerations. Because subscriptions incorporate an element of time, your app needs to have the appropriate logic to determine whether the subscription is currently active and what time periods the subscription was active in the past. Your app also needs to react to new and renewed subscriptions, and properly handle expired subscriptions. Figure 5-1 shows an example subscription timeline, including some of the complexities your app needs to handle.

Figure 5-1 Example subscription timeline



Calculating a Subscription's Active Period

Your app needs to determine what content the user has access to based on the period of time the subscription was active. Consider, for example, a magazine that publishes a new issue on the first day of each month, following the timeline in Figure 5-1.

On February 20, the user buys a one-month subscription. When the user buys the subscription, the issue that was published February 1 is delivered immediately. On March 20, the subscription is due to expire but automatically renews for another month. On April 20, the user chooses to let the subscription lapse. Finally, on May 17, the user resubscribes. At that time, the May issue is delivered immediately just as the February issue was for the original subscription.

A total of three issues are available to the user at the end of this example—February, March, and May—as shown in Table 5-1.

Table 5-1 Timeline of a sample subscription

Magazine issue	User has access to content?
January	No, subscription hasn't started
February	Yes, delivered when the user subscribed on February 20

Magazine issue	User has access to content?
March	Yes, delivered on March 1, at the time of its publication
April	No, subscription expired on May 20
May	Yes, delivered when the user resubscribed on May 17

To implement this logic in your app, keep a record of the date that each piece of content is published. Read the Original Purchase Date and Subscription Expiration Date field from each receipt entry to determine the start and end dates of the subscription. For information about the receipt, see *Receipt Validation Programming Guide* . The user has access to all content published between each start and end date as well as the content that was initially unlocked when the subscription was purchased.

If the subscription lapsed, there will be multiple periods of time during which the subscription was active, and there will be pieces of content unlocked at the beginning of a subscription period.

Note: Don't calculate the subscription period by adding a subscription duration to the purchase date. This approach fails to take into account the free trial period, the marketing opt-in period, and the content made available immediately after the user purchased the subscription.

Expiration and Renewal

The renewal process begins with a "preflight" check, starting ten days before the expiration date. During those ten days, the App Store checks for any issues that might delay or prevent the subscription from being automatically renewed—for example, if the customer no longer has an active payment method, if the product's price increased since the user bought the subscription, or if the product is no longer available. The App Store notifies users of any issue so that they can resolve it before the subscription needs to renew, ensuring their subscription isn't interrupted.

Note: Increasing the price of a subscription doesn't disable automatic renewal for all customers, only for those customers whose subscription expires in the next ten days. If this change is a mistake, changing it back to the original price means no additional users are affected. If this change is intentional, keeping the new higher price causes automatic renewal to be disabled for the rest of your users in turn as they enter the ten-day renewal window.

During the 24-hour period before the subscription expires, the App Store starts trying to automatically renew it. The App Store makes several attempts to automatically renew the subscription over a period of time but eventually stops if there are too many failed attempts.

The App Store renews the subscription slightly before it expires, to prevent any lapse in the subscription. However, lapses are still possible. For example, if the user's payment information is no longer valid, the first renewal attempt would fail. If the user doesn't update this information until after the subscription expires, there would be a short lapse in the subscription between the expiration date and the date that a subsequent automatic renewal succeeds. The user can also disable automatic renewal and intentionally let the subscription expire, then renew it at a later date, creating a longer lapse in the subscription. Make sure your app's subscription logic can handle lapses of various durations correctly.

After a subscription is successfully renewed, Store Kit adds a transaction for the renewal to the transaction queue. Your app checks the transaction queue on launch and handles the renewal the same way as any other transaction. Note that if your app is already running when the subscription renews, the transaction observer is *not* called; your app finds out about the renewal the next time it's launched.

Cancellation

A subscription is paid for in full when it's purchased and can be refunded only by contacting Apple customer service. For example, if the user accidentally buys the wrong product, customer support can cancel the subscription and issue a refund. It's not possible for customers to change their mind in the middle of a subscription period and decide they don't want to pay for the rest of the subscription.

To check whether a purchase has been canceled, look for the Cancellation Date field in the receipt. If the field has a date in it, regardless of the subscription's expiration date, the purchase has been canceled—treat a canceled receipt the same as if no purchase had ever been made.

Depending on the type of product, you may be able to check only the currently active subscription, or you may need to check all past subscriptions. For example, a magazine app would need to check all past subscriptions to determine which issues the user had access to.

Cross-Platform Considerations

Product identifiers are associated with a single app. Apps that have both an iOS and OS X version have separate products with separate product identifiers on each platform. You could let users who have a subscription in an iOS app access the content from an OS X app (or vice versa), but implementing that functionality is your responsibility. You would need some system for identifying users and keeping track of what content they've subscribed to, similar to what you would implement for an app that uses non-renewable subscriptions.

The Test Environment

For the sake of testing, there are some differences in behavior between auto-renewable subscriptions in the production environment and in the test environment.

Renewal happens at an accelerated rate, and auto-renewable subscriptions renew a maximum of six times per day. This lets you test how your app handles a subscription renewal, a subscription lapse, and a subscription history that includes gaps.

Because of the accelerated expiration and renewal rate, the subscription can expire before the system starts trying to renew the subscription, leaving a small lapse in the subscription period. Such lapses are also possible in production for a variety of reasons—make sure your app handles them correctly.

Restoring Purchased Products

Users restore transactions to maintain access to content they've already purchased. For example, when they upgrade to a new phone, they don't lose all of the items they purchased on the old phone. Include some mechanism in your app to let the user restore their purchases, such as a Restore Purchases button. Restoring purchases prompts for the user's App Store credentials, which interrupts the flow of your app: because of this, don't automatically restore purchases, especially not every time your app is launched.

In most cases, all your app needs to do is refresh its receipt and deliver the products in its receipt. The refreshed receipt contains a record of the user's purchases in this app, on this device or any other device. However, some apps need to take an alternate approach for one of the following reasons:

- If you use Apple-hosted content, restoring completed transactions gives your app the transaction objects it uses to download the content.
- If you need to support versions of iOS earlier than iOS 7, where the app receipt isn't available, restore completed transactions instead.
- If your app uses non-renewing subscriptions, your app is responsible for the restoration process.

Refreshing the receipt asks the App Store for the latest copy of the receipt. Refreshing a receipt does not create any new transactions. Although you should avoid refreshing multiple times in a row, this action would have same result as refreshing it just once.

Restoring completed transactions creates a new transaction for every completed transaction the user made, essentially replaying history for your transaction queue observer. While transactions are being restored, your app maintains its own state to keep track of why it's restoring completed transactions and how it needs to handle them. Restoring multiple times creates multiple restored transactions for each completed transaction.

Note: If the user attempts to purchase a product that'as already been purchased, rather than using your app's restoration interface, the App Store creates a regular transaction instead of a restore transaction. The user isn't charged again for the product. Treat these transactions the exact same way you treated the original transactions.

Give the user an appropriate level of control over what content is redownloaded. For example, don't download three years worth of daily newpapers or hundreds of megabytes worth of game levels all at once.

Refreshing the App Receipt

Create a receipt refresh request, set a delegate, and start the request. The request supports optional properties for obtaining receipts in various states during testing such as expired receipts—for details, see the values for the `initWithReceiptProperties:` method of `SKReceiptRefreshRequest`.

```
request = [[SKReceiptRefreshRequest alloc] init];
request.delegate = self;
[request start];
```

After the receipt is refreshed, examine it and deliver any products that were added.

Restoring Completed Transactions

Your app starts the process by calling the `restoreCompletedTransactions` method of `SKPaymentQueue`. This sends a request to the App Store to restore all of your app's completed transactions. If your app sets a value for the `applicationUsername` property of its payment requests, as described in "Detecting Irregular Activity" (page 21), use the `restoreCompletedTransactionsWithApplicationUsername:` method to provide the same information when restoring transactions.

The App Store generates a new transaction for each transaction that was previously completed. The restored transaction has a reference to the original transaction: instances of `SKPaymentTransaction` have a `originalTransaction` property, and the entries in the receipt have an Original Transaction Identifier field.

Note: The date fields have slightly different meanings for restored purchases. For details, see the Purchase Date and Original Purchase Date fields in *Receipt Validation Programming Guide*.

Your transaction queue observer is called with a status of `SKPaymentTransactionStateRestored` for each restored transaction, as described in "Waiting for the App Store to Process Transactions" (page 23). The action you take at this point depends on the design of your app.

- If your app uses the app receipt and doesn't have Apple-hosted content, this code isn't needed because your app doesn't restore completed transactions. Finish any restored transactions immediately.
- If your app uses the app receipt and has Apple-hosted content, let the user select which products to restore before starting the restoration process. During restoration, re-download the user-selected content and finish any other transactions immediately.

```
NSMutableArray *productIDsToRestore = <# From the user #>;
```

```
SKPaymentTransaction *transaction = <# Current transaction #>;


if ([productIDsToRestore containsObject:transaction.transactionIdentifier])
 {
    // Re-download the Apple-hosted content, then finish the transaction
    // and remove the product identifier from the array of product IDs.
} else {
    [[SKPaymentQueue defaultQueue] finishTransaction:transaction;]
}
```

- If your app doesn't use the app receipt, it examines all completed transactions as they're restored. It uses a similar code path to the original purchase logic to make the product available and then finishes the transaction.

  Apps with more than a few products, especially products with associated content, let the user select which products to restore instead of restoring everything all at once. These apps keep track of which completed transactions need to be processed as they're restored and which transactions can be ignored by finishing them immediately.

Preparing for App Review

After you finish testing, you're ready to submit your app for review. This chapter highlights a few tips to help you through the review process.

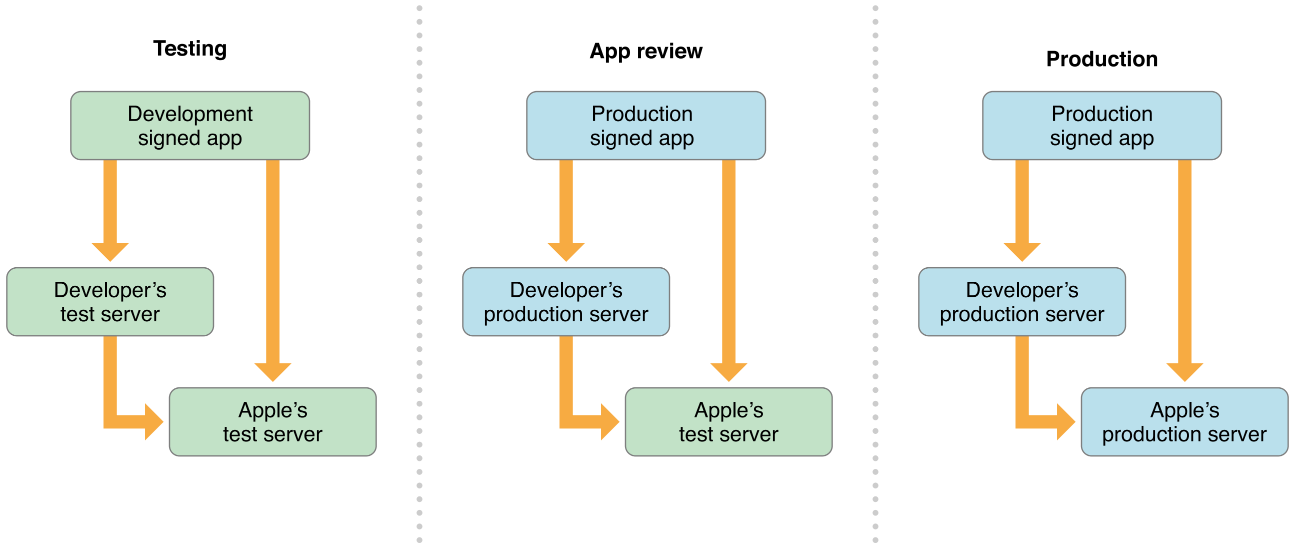
Submitting Products for Review

The first time you submit your app for review, you also need to submit in-app products to be reviewed at the same time. After the first submission, you can submit updates to your app and products for review independently of each other. For more information, see "In-App Purchase" in *iTunes Connect Developer Guide* .

Receipts in the Test Environment

Your app runs different environments while in development, review, and production, as show in Figure 7-1.

Figure 7-1 Development, review, and production environments



During development, you run a development-signed version of your app, which connects to your development servers and the test environment for the App Store. In production, your users run a production-signed version of your app which connects to your production servers and the production App Store. However, during app review, your app runs in a mixed production/test environment: it's production signed and connects to your production servers, but it connects to the test environment for the App Store.

When validating receipts on your server, your server needs to be able to handle a production-signed app getting its receipts from Apple's test environment. The recommended approach is for your production server to always validate receipts against the production App Store first. If validation fails with the error code "Sandbox receipt used in production", validate against the test environment instead.

Implementation Checklist

Before you submit your app for review, verify that you've implemented all of the required behavior. Make sure you've implemented the following core In-App Purchase behavior (listed in order of a typical development process):

- Create and configure products in iTunes Connect.

  You can change your products throughout the process, but you need at least one product configured before you can test any code.

- Get a list of product identifiers, either from the app bundle or your own server. Send that list to the App Store using an instance of `SKProductsRequest`.

- Implement a user interface for your app's store, using the instances of `SKProduct` returned by the App Store. Start with a simple inteface during development, such as a table view or a few buttons.

  Implement a final user interface for your app's store at whatever point makes sense in your development process.

- Request payment by adding an instance of `SKPayment` to the transaction queue using the `addPayment:` method of `SKPaymentQueue`.

- Implement a transaction queue observer, starting with the `paymentQueue:updatedTransactions:` method.

  Implement the other methods in the `SKPaymentTransactionObserver` protocol at whatever point makes sense in your development process.

- Deliver the purchased product by making a persistent record of the purchase for future launches, downloading any associated content, and finally calling the `finishTransaction:` method of `SKPaymentQueue`.

During development, you can implement a trivial version of this code at first—for example, simply displaying "Product Delivered" on the screen—and then implement the real version at whatever point makes sense in your development process.

If your app sells non-consumable items, auto-renewable subscriptions, or non-renewing subscriptions, verify that you've implemented the following restoration logic:

- Provide UI to begin the restoration process.
- Retrieve information about past purchases by either refreshing the app receipt using the `SKReceiptRefreshRequest` class or restoring completed transactions using the `restoreCompletedTransactions` method of the `SKPaymentQueue` class.
- Let the user re-download content.

  If you use Apple-hosted content, restore completed transactions and use the transaction's `downloads` property to get an instance of `SKDownload`.

  If you host your own content, make the appropriate calls to your server.

If your app sells auto-renewable or non-renewing subscriptions, verify that you've implemented the following subscription logic:

- Handle a newly-purchased subscription by delivering the most recently published piece of content—for example, the latest issue of a magazine.
- When new content is published, make it available to the user.
- When a subscription expires, let the user renew it.

  If your app sells auto-renewable subscriptions, let the App Store handle this process. Don't try to handle it yourself.

  If your app sells non-renewing subscriptions, your app is responsible for this process.
- When a subscription becomes inactive, stop making new content available. Update your interface so the user has the option to purchase the subscription again, re-activating it.
- Implement some system to keep track of when content was published. Use this system when restoring purchases to give the user access to the content that was paid for, based on the periods of time the subscription was active.

Document Revision History

This table describes the changes to *In-App Purchase Programming Guide*.

Date	Notes
2013-10-22	Expanded discussion of delivering products. Added a chapter on restoration. Minor changes throughout. Added discussion of `applicationUsername` property in "Requesting Payment" (page 20). Expanded discussion of persisting purchases and downloading content in "Delivering Products" (page 23). Added chapter "Restoring Purchased Products" (page 38). Added an implementation checklist in "Preparing for App Review" (page 41).
2013-09-18	Expanded and reorganized content throughout. Moved information about validating receipts to *Receipt Validation Programming Guide*.
2012-09-19	Removed note that expires_date key was not present on restored transactions. Best practice for restoring auto-renewable subscriptions is to simply respect the expires_date key on the restored transactions. Removed section on restoring auto-renewable subscriptions that indicated otherwise.
2012-02-16	Updated artwork throughout to reflect cross-platform availability. Updated code listing to remove deprecated method. Replaced the deprecated `paymentWithProductIdentifier:` method with the `paymentWithProduct:` method in "Adding a Store to Your Application".
2012-01-09	Minor updates for using this technology on OS X.
2011-10-12	Added information about a new type of purchase to the overview.

Date	Notes
2011-06-06	First release of this document for OS X.
2011-05-26	Updated to reflect the latest server behavior for auto-renewable subscriptions.
2011-03-08	Corrected the list of keys in the renewable subscriptions chapter.
2011-02-15	Apps must always retrieve product information before submitting a purchase; this ensures that the item has been marked for sale. Information added about renewable subscriptions.
2010-09-01	Minor edits.
2010-06-14	Minor clarifications to SKRequest.
2010-01-20	Fixed a typo in the JSON receipt object.
2009-11-12	Receipt data must be base64 encoded before being passed to the validation server. Other minor updates.
2009-09-09	Revised introductory chapter. Clarified usage of receipt data. Recommended the iTunes Connect Developer Guide as the primary source of information about creating product metadata. Renamed from "Store Kit Programming Guide" to "In App Purchase Programming Guide".
2009-06-12	Revised to include discussion on retrieving price information from the Apple App Store as well as validating receipts with the store.
2009-03-13	New document that describes how to use the StoreKit API to implement a store with your application.